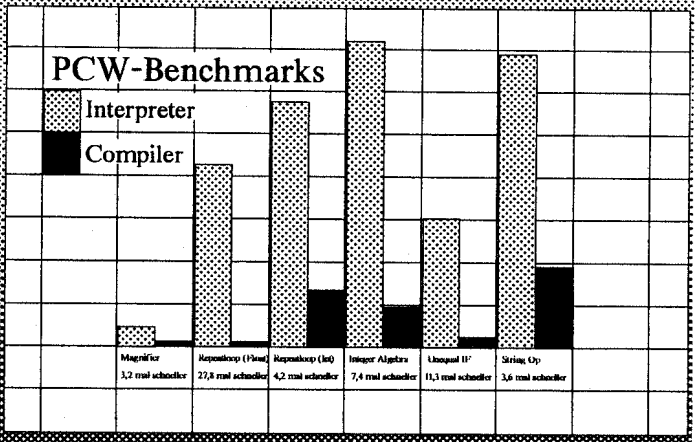
OMIKRON.

Junior-COMPILER

ATARI ST



... für den anspruchsvollen Programmierer

1. 1. Benutzungsbestimmungen

1.1 Copyright

Den OMIKRON.Compiler dürfen Sie zum eigenen Bedarf beliebig kopieren. Sie können ihn auf die Harddisk kopieren und auf jeder Arbeitsdiskette unterbringen. Stellen Sie jedoch bitte sicher, daß weder Kopien noch das Original in die Hände Dritter gelangen; schließlich haften Sie für eventuell entstehende Raubkopien.

Die Compiler-Library ("BASLIB" auf der Diskette) dürfen Sie im Zusammenhang mit Programmen verwenden bzw. weitergeben, die Sie mit dem OMIKRON.Compiler erstellt haben.

1.1.1 Copyright an Compilaten

Mit dem OMIKRON.Compiler erstellte Programme dürfen Sie in beliebiger Anzahl verwenden, verbreiten und verkaufen. Es sind keinerlei Lizenzgebühren an OMIKRON.Soft+Hardware GmbH abzuführen.

Eine Ausnahme: Interpreter für Allround-Programmiersprachen (wie BASIC oder PASCAL) dürfen Sie nur mit schriftlicher Genehmigung der OMIKRON.Soft&Hardware GmbH in Umlauf bringen. Für Compiler beliebiger Sprachen gilt diese Einschränkung nicht.

Teile des OMIKRON.BASIC-Interpreters bzw. der BASLIB dürfen jedoch nicht in das vom selbstgeschriebenen Compiler erzeugte Compilat eingebunden werden.

1.1.2 Hinweis auf OMIKRON.BASIC

In jedes mit dem OMIKRON.BASIC-Compiler erstellte Programm müssen Sie einen Hinweis auf die Verwendung von OMIKRON.BASIC unterbringen. Der Hinweis muß dem Benutzer des Programms zugänglich sein; automatisch erscheinen muß er jedoch nicht.

Beispiele:

-In der Info-Box (Box im Desktop-Info)

-im Titelbild

-in der Anleitung

Ein Hinweis genügt - Sie brauchen den Text nicht mehrfach unterzubringen. Ebenso genügt die kleinste Schrift (6 Punkte hoch), sie darf auch senkrecht stehen.

2. INHALTSVERZEICHNIS

Und noch eine Bitte, mit der Sie uns einen GROSSEN Gefallen tun können: **BITTE** rufen Sie uns wegen einem Programm, das compiliert nicht mehr läuft, **ERST** an, wenn Sie das Kapitel **'Unterschiede zum Interpreter'** sorgfältig durchgearbeitet haben.

3. EINLEITUNG

Wenn Sie auf Geschwindigkeit angewiesen sind, möchten wir Ihnen zum Kauf des OMIKRON.Compilers gratulieren: Sie haben soeben eine der schnellsten Hochsprachen erworben, die es für den ATARI ST überhaupt gibt.

3.1 Warum sind compilierte Programme eigentlich schneller?

Programme in BASIC muß der 68000-Prozessor bei jedem Ablauf erst mit Hilfe des BASIC- Interpreters dekodieren. Dieses Dekodieren kostet Zeit.

Maschinensprache ist die "Muttersprache" des 68000-Prozessors, der im ATARI ST eingebaut ist. Da der Prozessor Programme in Maschinensprache nicht erst bei jedem Ablauf dekodieren muß, laufen sie wesentlich schneller ab. Der OMIKRON.COMPILER übersetzt nun Programme, die Sie in OMIKRON.BASIC geschrieben haben, in Maschinensprache.

Wenn Sie also ein "normales" BASIC-Programm mit dem Compiler übersetzen, kann der Prozessor es direkt verstehen: es läuft schneller ab. Dagegen wird das Ausdrucken auf dem (langsamen) Drucker nicht schneller: schon im (relativ!) langsamen BASIC wartet der Computer die ganze Zeit auf den Drucker; im compilierten Programm ist der Computer noch etwas schneller fertig mit dem Bereitstellen der Zeichen, wartet aber dafür auch noch etwas länger auf den Drucker.

Eigentlich sollte ein Compiler ohne ein Handbuch auskommen. Der "ideale Compiler" übersetzt einfach ein bestehendes BASIC-Programm in Maschinensprache - fertig.

Leider gibt es den "idealen Compiler" nicht. Der OMIKRON.Compiler ist sehr einfach zu bedienen und kommt dem Ideal schon recht nahe - aber es gibt eben doch noch das eine oder andere zu beachten.

Die wichtigsten Einschränkungen des OMIKRON.Compilers sind die **"Unterschiede zum Interpreter"**. Leider ist es aus technischen Gründen nicht machbar, einen Compiler zu konstruieren, der alle BASIC-Programme ohne Änderung übersetzt. Einige Befehle bzw. einige spezielle Konstruktionenkönnen nicht übersetzt werden (Beispiel: EXIT 3 kann vom Interpreter verarbeitet werden. Der Compiler kann es jedoch nicht übersetzen). Alle diese "Unterschiede zum Interpreter" machen es erforderlich, daß Sie als Anwender ein neu zu schreibendes BASIC-Programm, das Sie später compilieren wollen, gleich ohne die kritischen Befehle schreiben; bzw. daß Sie bei bestehenden Programmen das Programm so ändern, daß es ohne diese Befehle oder Konstrukte auskommt.

Zum Glück ist es beim OMIKRON.Compiler so, daß die Liste der "Unterschiede zum Interpreter" sehr kurz ist und außerdem keine wichtigen Befehle enthält. 99% der BASIC- Programme lassen sich also mit keinen oder nur minimalsten Änderungen problemlos compilieren.

3.2 Der FPU-Compiler im Vergleich zum normalen Compiler

Ab der Version 3.5 kann Ihr Compiler Maschinensprache erzeugen, die die in Ihrem Rechner eventuell eingebaute FPU 68881 direkt anspricht. Nur wenn Sie eine solche FPU eingebaut haben, bringt Ihnen die FPU-Option irgendwelche Vorteile!

Sie können nun durch ein COMPILER-Steuerwort (siehe Kapitel "Compiler-Steuerworte") wählen, ob die Fließkommaberechnungen von der normalen CPU (wie gewöhnlich, und wie auch im Interpreter) oder aber von der FPU erledigt werden sollen. Wählen Sie CPU, erzeugt der Compiler eine auf allen ATARI ST/TT lauffähiges Programm. Wählen Sie die FPU-Option so werden automatisch alle Fließkommaberechnungen von der FPU ausgeführt (ohne Änderungen am Quelltext). Dabei sind Geschwindkeitssteigerungen im Bereich zwischen Faktor 10 und 100 möglich!

4. Die Bedienung des Compilers

Bitte kopieren Sie sich Ihre Originaldiskette auf eine Leerdiskette. Verwenden Sie ab dann nur noch die Kopie zum Arbeiten mit dem OMIKRON.Compiler. Ihre Originaldiskette sollten Sie sicher verwahren.

4.1 Start des Compilers vom Interpreter

Um mit der Bedienung des Compilers vertraut zu werden, sollten Sie das folgende Beispiel einmal ausprobieren.

1. Laden Sie den OMIKRON.BASIC Interpreter.

2. Schreiben Sie ein BASIC-Programm.

```
T=TIMER
FOR I=1 TO 10000
  A=A+I
NEXT I
PRINT A
PRINT "Zeit: "; ( TIMER-T)/200
STOP
```

3. Speichern Sie das fertige BASIC-Programm mit **SAVE "TEST.BAS"** auf Diskette (oder Harddisk, oder RAMDISK) ab. Sie sollten das BASIC-Programm, mit dem Sie arbeiten, aus prinzipiellen Erwägungen abspeichern. Für die Funktion des Compilers ist das nicht nötig.

4. Wählen Sie nun im Menü RUN des Editors in OMIKRON.BASIC den Menüpunkt "**COMPILE**" an.

Was nun passiert, ist folgendes:

Ihr OMIKRON.BASIC lädt den Compiler. "COMPILER.PRG" wird immer von Laufwerk C oder - falls C nicht vorhanden - von Laufwerk A aus dem Hauptverzeichnis geladen.

Konnte Ihr OMIKRON.BASIC den Compiler nicht finden, meldet es keinen Fehler. Dann bewirkt das Anklicken des Menüpunkts "COMPILE" einfach gar nichts. Stellen Sie daher sicher, daß sich der Compiler auf dem oben angegebenen Pfad befindet.

Das OMIKRON.BASIC startet nun den Compiler direkt. Der Compiler übersetzt dann das aktuelle BASIC-Programm, das Sie gerade im OMIKRON.BASIC-Interpreter bearbeiten (also in unserem Beispiel das gerade eingetippte "TEST.BAS").

Der Compiler arbeitet jetzt eine sehr kurze Zeit. In dieser Zeit übersetzt er unser Beispielprogramm in Maschinensprache. Danach speichert er das fertige Maschinenspracheprogramm auf demselben Pfad wieder ab, von dem das ursprüngliche .BAS-Programm geladen wurde.

Der Name des Programms ist derselbe wie der des BASIC-Programms, nur mit der Namenserweiterung (neudeutsch Extension) .PRG statt .BAS. In unserem Beispiel heißt Ihr BASIC-Programm "TEST.BAS". Daher heißt das in Maschinensprache übersetzte Programm nachher "TEST.PRG". Der Name Ihres BASIC-Programms wird im Editor des BASIC-Interpreters übrigens oben rechts angezeigt.

Das compilierte Programm können Sie jetzt vom BASIC-Interpreter aus aufrufen. Dazu klicken Sie einfach den Menüpunkt "EXEC *.PRG" aus dem Menü "RUN" an; und wählen das auszuführende Maschinensprache-Programm: in unserem Beispiel also TEST.PRG.

Weil unser Programm die Rechenzeit ausgibt, können Sie schön die Unterschiede in der Laufzeit zwischen dem ursprünglichen BASIC-Programm und dem compilierten Programm sehen (das compilierte Programm ist etwa 12mal so schnell).

Die gerade beschriebene Art, den Compiler zu bedienen, ist die praktischste und effizienteste Art. Sie brauchen den Editor des Interpreters nicht zu verlassen und können so am schnellsten entwickeln.

Der Vollständigkeit halber sei jedoch noch die andere Art erwähnt, den Compiler zu bedienen:

4.2 Start des Compilers vom Desktop

Starten Sie vom Desktop aus COMPILER.PRG (durch Doppelklick)

Der Compiler stellt nun eine Dateiauswahlbox dar. Wählen Sie nun das zu compilierende BASIC- Programm. Das Programm muß mit SAVE (nicht mit SAVE,A) abgespeichert worden sein!

Der Compiler lädt nun das Programm, übersetzt es in Maschinensprache und speichert es mit der Extension .PRG auf demselben Pfad, auf dem auch das .BAS war, wieder ab. Beispiel: Sie wählen C:\BASIC\HILBERT.BAS zum Compilieren an. Dann wird der OMIKRON.Compiler das compilierte Programm unter dem Namen HILBERT.PRG auf Laufwerk C im Ordner BASIC wieder ablegen (C:\BASIC\HILBERT.PRG).

Sobald der Compiler fertig ist, stellt er die Dateiauswahlbox wieder dar. Sie können dann ein weiteres BASIC-Programm compilieren.

Wenn Sie kein weiteres Programm mehr compilieren wollen, verlassen Sie den Compiler, indem Sie in der Dateiauswahlbox auf den Abbruch-Knopf klicken.

5. Die Ausgaben des Compilers während des Compiliervorgangs

Während der Compiler Ihr BASIC-Programm übersetzt, gibt er einige Meldungen über den Verlauf des Compiliervorgangs auf den Bildschirm aus. Er protokolliert sozusagen seine Arbeit.

Oben auf dem Bildschirm sehen Sie die Copyrightmeldung des Compilers. Aus dieser erfahren Sie die Versions- und Seriennummer. Darunter gibt der Compiler das eigentliche Protokoll aus. In jedem Fall gibt er aus:

```
COMPILING DATA/UNLINKING UNUSED LIBRARY STATE-
MENTS
```

Der Compiler prüft in diesem ersten Lauf das Programm auf unbenutzte Prozeduren und Funktionen. Befinden sich in Ihrem Programm Prozeduren oder Funktionen, die zwar definiert sind, jedoch nie aufgerufen werden, wird eine entsprechende Warnmeldung ausgegeben (siehe unten). Sind diese Prozeduren Teil einer Library, die mit dem LIBRARY-Befehl in das BASIC-Programm eingeladen wurde, so werden alle diese unbenutzten Teile ausgespart. In einem compilerten Programm sind also immer nur die benutzten Teile einer Library vorhanden, der Rest wird sozusagen weggeschnitten.

Beim normalen Programm kürzt der Compiler nichts automatisch heraus: er übersetzt immer das gesamte Programm.

Wird eine unbenutzte Prozedur gefunden, dann erscheint auf dem Bildschirm direkt unter **COMPILING DATA/UNLINKING UNUSED LIBRARY STATEMENTS** die Warnmeldung **UNDEFINED STATEMENTS OR DIM'S:**. Angenommen, eine Prozedur AUSGABE würde in Ihrem Programm aufgerufen, wäre aber nirgendwo mit DEF PROC AUSGABE definiert. Der Compiler würde dann ausgeben:

```
UNDEFINED STATEMENTS OR DIM'S: PROC AUSGABE
```

Ein sehr wichtiger Unterschied zum Interpreter betrifft den DIM-Befehl bzw. die Möglichkeit, Feldvariable ohne vorherige Dimensionierung zu verwenden. Im Interpreter können Sie Feldvariable (z.B. A(7)) bis zu einem maximalen Index von 10 ohne vorheriges DIM verwenden. Im Compiler ist das nicht möglich!

Sie müssen also jede verwendete Feldvariable vorher mit DIM dimensionieren, auch wenn das im Interpreter nicht erforderlich sein sollte!

Wenn Sie in Ihrem Programm eine Feldvariable ohne vorherige DIMensionierung verwenden, stellt der Compiler das fest und gibt z.B. aus:

```
UNDEFINED STATEMENTS OR DIM'S: DIM A(.)
```

In einem solchen Fall fügen Sie in Ihrem BASIC-Quelltext einfach das fehlende DIM ein und compilieren erneut.

Stellt der Compiler im ersten Durchlauf keine fatalen Fehler[1] fest , dann folgt der zweite und der dritte Durchlauf. Die Zeilennummer der gerade in Arbeit befindlichen Zeile wird auf dem Bildschirm dargestellt.

Der Compiler kann auch im zweiten oder dritten Durchlauf noch einige Fehlermeldungen oder Warnhinweise erzeugen. Welche das im einzelnen sind, sehen Sie bitte unter "Fehlermeldungen" in diesem Handbuch nach.

Drücken Sie während des Compiliervorgangs irgendeine Taste, dann wartet der Compiler nach Beendigung des Compilierens noch auf einen weiteren Tastendruck.

Sie können sich so die Meldungen des Compilers in aller Ruhe ansehen. Dasselbe gilt übrigens, wenn während des Compilierens eine Fehlermeldung oder eine Warnung ausgegeben wurde: Auch dann wartet der Compiler nach Beendigung des Compiliervorgangs noch darauf, daß Sie eine Taste drücken.

5.1 Das erzeugte Maschinensprache-Programm

Der OMIKRON.Compiler erzeugt aus Ihrem BASIC-Programm (Endung .BAS) ein Maschinenspracheprogramm (Endung .PRG).

Allerdings fehlen diesem Maschinenspracheprogramm noch einige wichtige Teile. Dies sind allgemeine Unterprogramme, die fast alle BASIC-Programme brauchen: die gesamte Fließkomma-Arithmetik, die Zeichenein- und -Ausgabe,

[1]fatale Fehler sind z.B.: ein DIM fehlt, oder eine Prozedur wird zwar aufgerufen, ist aber nicht definiert.

usw. Diese Unterprogramme finden sich in der Datei BASLIB, die auf der Compiler-Original-Diskette mit ausgeliefert wird.

Ein vom Compiler erzeugtes .PRG-Programm lädt, wenn es gestartet wird, automatisch die BASLIB nach.

5.2 Der Sinn der BASLIB - Fragen zum Konzept

Warum erzeugt der Compiler nicht ein Programm, das ohne die BASLIB auskommt?

Die Antwort ist: es braucht mehr Platz auf der Diskette. Die BASLIB ist etwa 40kByte lang. Wenn Sie nun 10 compilierte BASIC-Programme auf einer Diskette abspeichern würden, die alle die BASLIB bereits enthielten, hätten Sie 400kByte Speicherbedarf auf Ihrer Diskette. Speichern Sie die Programme jeweils ohne BASLIB und dazu eine BASLIB einzeln, dann haben Sie nur 40kByte Speicherbedarf.

Ein compiliertes Programm lädt die BASLIB also beim Starten nach. Von welchem Pfad? Was geschieht, wenn die BASLIB nicht gefunden wird?

Das compilierte Programm sucht die BASLIB zunächst in demselben Pfad, in dem das Programm selber auch steht. Steht das Programm beispielsweise im Ordner BASIC auf Laufwerk C, dann sucht es zunächst im Ordner BASIC auf Laufwerk C nach der BASLIB. Findet es sie dort nicht, sucht es im Hauptverzeichnis des aktuellen Laufwerks (hier also C). Ist auch dort keine BASLIB, sucht es in der folgenden Reihenfolge:

1. Hauptverzeichnisse der Laufwerke C bis P
2. das übergeordnete Verzeichnis des gestarteten Programms
3. Laufwerk A

Findet es die BASLIB nirgendwo, meldet es "CAN'T LOAD BASLIB" und bricht ab.

Es gibt eine Möglichkeit, Programme zu generieren, die nichts mehr nachladen (die also die BASLIB bereits enthalten). Wie das geht, schlagen Sie bitte bei CUTLIB im Kapitel "Mitgelieferte Programme" bzw. im Kapitel "Compiler-Steuerworte" nach.

6. Unterschiede zum Interpreter

6.1 Spezielle Unterschiede (auch gegenüber älteren Compilerversionen)

Aus Gründen der Kompatibilität sind in der neuen Version des Compilers einige Dinge ganz weggefallen. Dadurch wurde die uneingeschränkte Lauffähigkeit auf allen Systemen (ATARI ST oder TT, mit Beschleunigerkarten oder Großbildschirmen) sichergestellt. Von den Streichungen betroffen sind:

- **die Begrenzung die Textausgabe** (über PRINT) auf einen bestimmten Bereich. Die dafür vorgesehenen Steuerzeichen CHR$(4) und CHR$(5) bewirken nichts und werden direkt an das BIOS weitergereicht. Dieses - zugegebenermaßen - schöne Feature mußte fallen, da es auf Großbildschirmen nicht zu verwirklichen ist.

- der **SCREEN** Befehl. Er brachte ebenfalls erhebliche Schwierigkeiten auf Großbildschirmen und mußte deshalb wegfallen. Wer unbedingt mit mehreren Bildschirmen arbeiten muß, der kann uneingeschränkt mit den XBIOS-Funktionen arbeiten oder den Bildschirm über BITBLT Befehle umschalten.

- die OMIKRON.Basic **Sprites.** Sie sind ebenfalls aus Kompatibilität zum Großbildschirm weggefallen.

6.2 Grundsätzliche Unterschiede

- Stringvariablen, die in **FIELD**-Anweisungen verwendet werden, dürfen nicht lokal verwendet werden, bis die **FIELD**-Anweisung durch eine Zuweisung mit **LET** in die Stringvariable oder durch **CLOSE** aufgehoben wird, d.h., sie dürfen nicht **LOCAL** deklariert werden und nicht in **DEF PROC** oder **DEF FN** verwendet werden.

Beispiel **(verboten!!!)**:

```
FIELD 1,5 AS A$
AUSGABE(X$)
 .
END
'
DEF PROC AUSGABE(A$) ' FALSCH! Die FIELD-Va-
riable darf nicht lokal verwendet werden!
 PRINT A$
RETURN
```

-**GOTO <Stringausdruck>** berücksichtigt nur die ersten acht Zeichen eines Labels. Weitere Zeichen danach werden ignoriert.

-Die Division einer Integer-Zahl durch eine weitere Integer-Zahl kann, wenn die Division aufgeht, im Interpreter ein Integer-Ergebnis liefern; im Compiler ergibt dies jedoch einen Fließkommawert. Eine Division zweier Integer-Zahlen, die immer aufgeht, sollte man daher besser mit der Integer-Division "\" berechnen. **A=A/2** schreibt man also möglichst **A=A\2**.

-Variablenfelder (Arrays) werden nicht automatisch **DIM**ensioniert. Undimensionierte Felder müssen Sie am Programmanfang bzw. nach jedem **CLEAR**-Befehl selbst dimensionieren.

-Bei Fehlern, die zur Verzweigung zu Ihrer eigenen Fehlerroutine führen (mit **ON ERROR GOSUB**), wird der Endwert von Integer-FOR-NEXT-Schleifen zerstört. Wenn Sie also innerhalb einer Integer-FOR-NEXT-Schleife einen Fehler haben, funktioniert RESUME NEXT nicht mehr richtig.

-Innerhalb von Prozeduren und mehrzeiligen Funktionen, die Arrayelemente als lokale Variablen bzw. Parameter verwenden (Beispiel: GEMLIB) dürfen keine Dateien geöffnet oder geschlossen bzw. keine Felder dimensioniert oder um-dimensioniert werden.

-Lokale Variablen werden nicht im Garbage-Segment, sondern im Stringsegment und auf dem Prozessorstapel gehalten. Sie müssen daher evtl. den Prozessorstapel vergrössern (**CLEAR ,X**).

-**EXIT** geht im Compiler nur als **EXIT** ohne Anzahl oder **EXIT -1**.

EXIT 3 geht also z.B. nicht.

EXIT TO ist nur zum Verlassen einer Schleife erlaubt; **EXIT TO** kann nicht zum Verlassen einer Prozedur oder Funktion genutzt werden.

EXIT -1 zerstört die globalen Werte lokaler Variablen (setzt keine lokalen Variablen zurück, sondern löscht nur den Stapel des Prozessors).

-**INPUT$** und **INPUT USING** werden durch **ON TIMER GOSUB** unterbrochen, selbst mit "Multitasking_Between_Statements" (dazu siehe unter "COMPILER-STEUERWORTE")

-**CALL** erfordert "Multitasking_Between_Statements".

-Programme, die die Befehle **ON ERROR, ON TIMER, ON MOUSEBUT, ON KEY** oder **ON HELP** enthalten, müssen in der Regel die beiden Compiler-Steuerworte "Multitasking_Between_Statements" und "Trace_On" enthalten (die Bedeutung der Steuerworte finden Sie im Kapitel "COMPILER-STEUERWORTE").

-Funktionen müssen den Typ im Namen tragen, von dem das Ergebnis der Funktion sein soll.

Soll Ihre selbstdefinierte Funktion z.B. Single-Float-Werte zurückgeben, so dürfen Sie das "!"-Postfix für Single-Float nicht vergessen. Bei Double-Float müssen Sie entsprechend ein "#" anhängen:

Richtig ist:

```
DEF FN Mwst!(Betrag!):RETURN Betrag!*1.14
```

Falsch wäre: `DEF FN Mwst(Betrag!)` denn hier fehlt das Postfix für Single-Float, das Ausrufezeichen.

Richtig ist:

```
DEF FN Umfang#(Radius)
    RETURN Radius*2*PI
```

Die Integervariable Radius stört hier nicht, da PI dafür sorgt, daß alles in Double-Float gerechnet wird! (siehe unter Kapitel "Verarbeitungstypen bei Funktionen").

Falsch ist z.B. auch:

`DEF FN A(X)=1/X:` 1/X ist eine Fließkommazahl. Sie wird nun fälschlicherweise auf Integer gerundet.

oder:

`DEF FN A#(X)=X*2+17`: eine Fließkommafunktion kann keinen Integerwert haben

Richtig wäre hier:

```
DEF FN A(X)=X*2+17
```

Sämtliche Schwierigkeiten bei mehrzeiligen Funktionen können Sie durch eine Typenumwandlung umgehen, z.B.:

```
DEF FN Umfang!(Radius): RETURN CSNG ( Radius * 2 *
PI)
```

Im Regelfall wird der OMIKRON.Compiler Sie auf falsche Typen hinweisen:

```
Warning: RETURN typemismatch
```

-Der bei **READ** erwartete Datentyp muß mit dem jeweils gelesenen Datenelement übereinstimmen.

Also nicht:

READ A!,B!,C!: DATA 1.5,3,4.5: 3 ist eine Integer-Zahl

sondern:

READ A!,B!,C!: DATA 1.5,3.,4.5: 3. ist eine Fließkomma-Zahl

-Variablen vom Typ %F (Flag) dürfen nicht lokal verwendet werden (bei LOCAL oder DEF PROC / DEF FN)

-Übergibt man eine Variable vom Typ %F an eine Prozedur, so erhält die Variable keinen Wert zurückgegeben, auch wenn in der Definition der Prozedur "R Variable" angegeben wurde.

-Terme werden in beliebiger Reihenfolge ausgewertet. Ein Term in der Form

FNA(X)+FNB(Y) mit
DEF FNA(X):Y=X^2:RETURN X^3
DEF FNB(Y):RETURN Y

ergibt also entweder **X^3+X^2** oder **X^3+Y**

-**ASC("")** ist null; es ergibt sich kein ? Illegal function call

-**DIM** kann auch Felder verkleinern

-2000000000+100000000 ergibt -1294967296. Beide Zahlen sind Integer-Zahlen, deshalb handelt es sich hier um eine Integer-Addition, deren Zahlbereich von -2147483648 bis 2147483647 gesprengt wurde. Im Interpreter wird dieser

Fehler erkannt und das Ergebnis (3000000000) als Fließkommazahl weiterverarbeitet. Für den Compiler wird jedoch ein bestimmter Typ als Verarbeitungstyp gewählt, der später nicht mehr geändert werden kann. In der Regel ergibt sich der Verarbeitungstyp aus dem Wert oder aus den Werten, die miteinander verrechnet werden. (siehe auch Kapitel "Verarbeitungstypen von Funktionen").

-Ein Überlauf (?Overflow) wird bei Fließkommaarithmetik immer, bei Integerarithmetik nie bemerkt.

Insbesondere bei Integer Word und Integer Byte kann ein Überlauf leicht auftreten.

7. VERARBEITUNGSTYPEN DER FUNKTIONEN

Das Ergebnis ist immer vom Typ:

String: @ UPPER$ LOWER$ ERR$ TIME$ DATE$ HEX$ STR$ OCT$ BIN$ INPUT$ INKEY$ CHR$ MID$ LEFT$ RIGHT$ STRING$ MIRROR$ MKD$ MKS$ MKIL$ MKI$ **SPC** SPACE$

Single Float: **RND** CSNG CVS

Float (single oder double): DET EXP LN LOG SQR FACT ^ / /2 SIN COS TAN COT ATN ARCSIN ARCCOS ARCTAN ARCCOT SINH COSH TANH COTH SECH COSECH ARSINH ARTANH ARCOTH SEC COSEC

Double Float: PI **VAL** CDBL CVD

Integer: CSRLIN ERR ERL MEMORY HIGH LOW SEGPTR **TIMER** MOUSEX MOUSEY MOUSEBUT EOF LOF LOC FRE LPOS POS USR PEEK WPEEK LPEEK INSTR LEN ASC VARPTR POINT CINTL CINT CVIL CVI BIT **SGN** NOT SHL SHR = < > <> >= <= **AND** OR IMP XOR EQV NAND NOR

vom Typ des Arguments: MIN MAX **INT** ABS **FIX FRAC** - * \ **MOD** + +1 -1 *2

Bei den in **fett** gesetzten Funktionen mag es vielleicht nicht immer sofort einleuchten, warum das Ergebnis vom angegebenen Typ ist. Passen Sie deshalb bei diesen Funktionen besonders auf, damit Sie die Typen nicht verwechseln.

Beispiel: INT(333333333333333.333) ist 333333333333333 (in double float gerechnet, da das Long-Integer-Format gar nicht ausreichen würde)

8. COMPILER-STEUERWORTE

Achtung! Für die folgende Art der Verwendung von Compiler-Steuerworten ist ein Interpreter der Version 3.x erforderlich!

COMPILER-STEUERWORTE beeinflussen den Ablauf des Compiliervorgangs, sie bewirken eine Steuerung des Compilers (bzw. des Compilats) durch das zu compilierende Programm.

Normalerweise übersetzt der Compiler das BASIC-Programm Befehl für Befehl in Maschinensprache. Doch von dieser Regel gibt es Ausnahmen: man kann in das BASIC-Programm Steuerworte für den Compiler hineinschreiben, die ihn zu irgendeiner Aktion veranlassen. Wohlgemerkt: sie veranlassen den Compiler beim Compilieren zu irgendeiner Aktion - sie haben nichts mit dem Ablauf des Programms im Interpreter zu tun! Der Interpreter behandelt die Compiler-Steuerworte wie REM-Anweisungen - er tut einfach gar nichts.

Alle Compiler-Steuerworte beginnen mit dem BASIC-Befehl **"COMPILER"**. Ihr OMIKRON.BASIC- Interpreter versteht den Befehl COMPILER. Der Interpreter beachtet eine solche Anweisung überhaupt nicht, sie wird einfach übergangen.

Anders hingegen beim Compiler. Wenn dieser beim Übersetzen auf eine COMPILER-Anweisung trifft, wird der folgende Ausdruck bewertet und als Steuerwort interpretiert.

Ein einfaches Beispiel zeigt die Benutzung eines Steuerwortes:

Sie schreiben irgendwo (am besten am Anfang) in Ihr Basic-programm: **COMPILER "CTRL_C OFF"**

Dies hat zur Folge, daß das compilierte Programm später nicht mehr durch Control-C unterbrochen werden kann. Der Compiler wird also angewiesen, bei der Programmübersetzung die Abfrage der Control-C Taste wegzulassen.

Im Folgenden sollen nun alle COMPILER-Steuerworte und deren Auswirkungen beschrieben werden.

COMPILER "TRACE ON" oder **COMPILER "TRON":**

Vor jedem Befehl wird auf Tastatur-, Timer- und Mausunterbrechungen geprüft. Dadurch wird Multitasking (ON TIMER, ON HELP, ON KEY oder ON MOUSEBUT) zwschen den Befehlen ermöglicht.

Zusätzlich wird nach jeder Zeile der aktuelle Programm- und Stapelzeiger festgehalten. Nur dadurch ist es im Fehlerfall möglich die Zeile, in der der Fehler auftrat, überhaupt festzustellen. Dies ist für zwei Dinge wichtig:

1. Für die Ausgabe einer Fehlermeldung! **Die wirkliche Fehlerzeile wird nur bei TRACE ON ermittelt.**

2. Für eigene Fehlerbehandlungen. Will man Programmfehler selbst mittels **ON ERROR GOSUB** behandeln, um mit RESUME auch die Fehlerstelle wiederzufinden, so muß TRACE ON eingestellt sein.

COMPILER "TRACE OFF" oder **COMPILER "TROFF":**

Es wird nicht mehr auf Unterbrechungen überprüft und auch der Programm- und Stapelzeiger wird nicht nach jeder Zeile festgehalten. Dadurch ist Multitasking nur noch mit gesetztem "MULTITASKING ALWAYS" (siehe unten) möglich, Fehlermeldungen ermitteln keine Zeilennummer und eigene Fehlerbehandlungen, die auch RESUME benötigen, funktionieren nicht.

COMPILER "MULTITASKING ALWAYS" oder **COMPILER "MA":**

Als weiter über die Möglichkeiten des Interprters hinausgehende Fähigkeit unterstützt der Compiler "richtiges" Multitasking. D.h. alle Unterbrechungsanforderungen werden nicht nur zwischen den Befehlen abgefragt, sondern stets mit einer eigenen Serviceroutine behandelt. Dadurch werden auch Befehle unterbrechbar, die eine lange Ausführungszeit benötigen (z.B. SORT oder INPUT).

WICHTIG: Die durch die Unterbrechung aufgerufenen Unterprogramme dürfen keine Stringverarbeitung enthalten. Wird ein eigenes Maschinenprogramm mit CALL aufgerufen, so sollte das Multitasking abgeschaltet sein.

Tip: Um eine INPUT USING Anweisung mit einem Mausklick zu unterbrechen, genügt es wenn die Unterbrechungroutine ein POKE RESERVED(4),1 ausführt. Darauf wird die INPUT USING Anweisung unmittelbar verlassen. Der Returnwert ist in diesem Fall -3.

COMPILER "MULTITASKING BETWEEN STATEMENTS oder **COMPILER "MBS":**

Keine zusätzlichen Unterbrechungen möglich, allenfalls Abfrage zwischen den Befehlen (falls TRACE ON). Diese Einstellung ist vor jedem **CALL** unbedingt erforderlich.

COMPILER "NO LINENUMBERS"

Um im Fehlerfall oder für RESUME (siehe auch TRACE ON) die Zugehörigkeit einer Programmstelle zu einer bestimmten Zeile festzustellen, wird normalerweise eine Tabelle angelegt. Diese Tabelle benötigt 6 Bytes pro Zeile und verlängert somit größere Programme um einige kByte.

Soll dieser Platz eingespart werden, kann man mit "NO LINENUMBERS" die Tabelle unterdrücken. Allerdings ist ohne diese Tabelle, das Ermitteln einer bestimmten Zeile unmöglich: **Fehler treten dann immer in Zeile 0 auf bzw. erzeugen die Meldung USE "TRACE ON" !**

COMPILER "LOAD INF"

Mit dieser Anweisung lädt das Compilat die "OM-BASIC.INF"-Datei, die verschiedene Einstellungen und vor allem Tastendefinitionen enthält.

Hinweis: in früheren Versionen wurde die INF-Datei immer geladen, obwohl deren Inhalt in den meisten Fällen für das Programm uninteressant war. Aus diesem Grund ist das Laden nunmehr optional.

COMPILER "SYMBOLS" oder COMPILER "SYMBOLS2"

Der Compiler erzeugt zusätzlich eine Symboltabelle, die bei der Fehlersuche an Compilaten nützlich sein kann. Es werden alle Zeilen, Labels, Prozeduren und Funktionen als Symbol in die Tabelle aufgenommen.

Tip: Damit die Fehlersuche am Compilaten reibungslos klappt, sollten Sie so vorgehen: Programm mit dem Debugger laden. Danach mit "GO , .start" und "GO , .ln_0" bis zur ersten Zeile vorrücken. Das Einzelschritt-Testen des ganzen Initteils davor ist nicht zu empfehlen, da hier die BASLIB nachgeladen und das Programm selbst verschoben werden muß. Das Steuerwort "NOEX" (siehe unten) muß bei der Verwendung eines Debuggers unbedingt mit angegeben werden. Auf der Diskette finden sie ein Beispielprogramm, welches sich mit einem Debugger starten läßt.

Wichtig: Wenn Sie den Debugger "BUGABOO" oder "OMDEBUG" verwenden, schreiben Sie statt "SYMBOLS" einfach "SYMBOLS2".

COMPILER "NO EXCEPTIONS" oder **COMPILER "NOEX"**:

Es werden keine eigenen Exception-Handler eingebunden. Bei einem entsprechenden Fehler stürzt das Compilat mit "Bomben" ab. Bei der Programmierung von Accessories sollten Sie die Exception-Handler stets abschalten. Es ist sonst möglich, daß durch den Absturz der Hauptapplikation auch das Accessory beendet wird, was das System zum endgültigen Absturz bringt. Sehr wichtig ist dieses Steuerwort auch bei der Fehlersuche mit einem Debugger. Nur wenn das Compilat die Fehler nicht selber abfängt hat der Debugger überhaupt eine Chance den Fehler richtig anzuzeigen.

COMPILER "FPU"

Der Compiler wird angewiesen ein Compilat für eine IO-FPU zu erzeugen (nicht eine Coprozessor-FPU wie im ATARI TT). So übersetzte Programme nutzen bei allen Fließkommaberechungen automatisch die Rechenkapazität des FPU-Prozessors aus, was einen Geschwindigkeitszuwachs bei solchen Operationen zwischen Faktor 10 und 100 ermöglicht. So übersetzte Programme sind auf einem ATARI ST ohne IO-FPU nicht lauffähig.

COMPILER "FPU ERRORS ON" bzw **COMPILER "FPU ERRORS OFF"**

Diese beiden Steuerworte schalten die Fehlerabfrage bei FPU Fehlern ein oder aus (nur bei Compilaten für IO-FPU relevant). Solche Fehler können sein: Teilen durch Null, Überlauf, oder negatives Argument bei Logarithmus. Ist

die Fehlerabfrage abgeschaltet bleiben Fehler unerkannt und es wird mit eventuell falschen Werten weitergerechnet.

COMPILER "BAS_MEM X"

Diese Einstellung teilt dem Compiler die für das Minimal-Speichermodell (siehe Kapitel Speicheraufbau) benötigte Menge an Basic Speicher mit. "X" steht dabei für die Zahl der benötigten Bytes.

COMPILER "STACK X"

... ist die entsprechende Einstellung für die Stackgrösse im Minimal-Speichermodell (siehe Kapitel Speicheraufbau).

COMPILER "WARNINGS OFF"

... unterdrückt die Bildschirmausgabe der "WARNINGS".

COMPILER "CUTLIB -X"

... bewirkt das automatische Aufrufen des Linkers "CUTLIB.PRG" nach dem erfolgreichen Compiliervorgang. "CUTLIB.PRG" sollte auf dem gleichen Pfad wie der COMPILER selbst zu finden sein. "-X" steht für beliebige Optionen die in der Kommadozeile an den Linker übergeben werden können.

COMPILER ON und **COMPILER OFF**

Das Steuerwort COMPILER OFF veranlaßt den Compiler, das Programm ab dieser Stelle nicht mehr weiter zu übersetzen. Erst wenn er im weiteren Verlauf des Programms wieder auf ein COMPILER ON trifft, wird die Programmüberstzung fortgesetzt. Es ist durch diese Steuerworte möglich bestimmte Programmteile nur im Interpreter zu benutzen (z.B. Testfunktionen) oder ganz bewußt bestimmte Teile eines Programm vorübergehend einfach wegzulassen (z.B. für eine abgemagerte Demoversion).

Im OMIKRON.BASIC gibt es ab Version 3.00 die (undokumentierten) Befehle **MEMORY_MOVE** und **MEMORY_MOVEB**. Bei überlappenden Bereichen ist es wichtig, ob vorwärts oder rückwärts kopiert werden soll. Das können Sie mit den folgenden beiden Steuerworten einstellen:

COMPILER "MEMORY MOVE FORWARD" oder COMPILER "MMF";

Blöcke werden von vorne nach hinten kopiert. Dies ist die Default-Einstellung.

COMPILER "MEMORY MOVE BACK" oder COMPILER "MMB"

Blöcke werden nun von hinten nach vorne kopiert.

Der Befehl **COMPILER** ist dem OMIKRON.BASIC-Interpreter erst ab Version 3.00 bekannt. Sollten Sie eine ältere Version besitzen, können Sie eine Version 3 beim ATARI-Fachhändler für 20 DM erwerben. Eine Version 3 auf die neueste Version 3.xx upzudaten, ist nur über die Diskette PD172 der Fa. Maxon (ist gleich Redaktion ST Computer), Postfach 5969, Industriestr. 26, 6236 Eschborn, möglich.

Für Besitzer früherer Interpreterversionen sind die Compilersteueranweisungen auch verwendbar. In diesem Fall schreiben Sie einfach die Steueranweisung (z.B. Trace_On) in Ihr Programm hinein:

Version 3.x des Interpreters:

```
PRINT I
COMPILER "Trace_On"
PRINT I*2
```

etc.

Version 2.x des Interpreters:

```
PRINT I
Trace_On
PRINT I*2
```

etc.

Für den Compiler sind diese beiden Versionen des Programmes gleich. Der Interpreter hingegen faßt das Trace_On im unteren Programm als Prozeduraufruf einer Prozedur namens Trace_On auf. Weil keine solche Prozedur vereinbart ist, gibt er ein ?Undef'd Statement aus. Sie als Benutzer müssen also eine Prozedur namens Trace_On definieren, die gar nichts tut:

```
DEF PROC Trace_On
 REM diese Prozedur macht gar nichts
RETURN
```

8.1 Die Systemvariable COMPILER

Wenn Ihr Programm im Interpreter anders ablaufen soll als im Compiler, erreichen Sie das ganz einfach, indem Sie die Systemvariable COMPILER abfragen. Beispiel:

```
IF COMPILER THEN
 PRINT "Programm läuft compiliert!"
ELSE
 PRINT "Programm läuft im Interpreter!"
ENDIF
```

Für Experten (und nur ab BASIC Version 3.xx):

```
RESERVED(Offset)
```

ergibt die Adresse eines Speicherbereichs mit untenstehendem Aufbau:

0: Adresse des Rückgabewertes bei CALL, der in D0 übergeben wird. Mit LPEEK(RESERVED (0)) können Sie also einen Wert vom Maschinenprogramm an das aufrufende BASIC-Programm übergeben.

4: Flag für INPUT USING

Falls Sie während eines INPUT USING-Befehl die Eingabe durch Timer- oder Mausinterrupt unterbrechen wollen, so setzen Sie in der Interruptroutine mit POKE RESERVED(4),1 ein Flag, das die Eingabe mit Rückgabewert -3 abbricht. Der Interpreter beendet die Eingabe ebenfalls, wenn ein "M" im Steuerstring vorkommt.

VERSION

Die Systemvariable VERSION ist im Compilat größer als die größtmögliche Versionsnummer im Interpreter (z.Zt. 303 für Version 3.03)

Dadurch werden eventuell im Interpreter eingebaute Versionsabfragen im compilierten Progamm übergangen.

```
IF VERSION < 300 THEN
   PROC Patch_Basic
ENDIF
```

9. Fehlermeldungen

Manche der unter "Unterschiede zum Interpreter" genannten Dinge (zum Beispiel nicht DIMensionierte Variablenfelder) erkennt der OMIKRON.Compiler automatisch beim Compilieren. In solchen Fällen gibt er während des Compiliervorgangs eine Fehlermeldung aus. In diesem Kapitel geben wir Ihnen eine komplette Übersicht über alle möglichen Fehlermeldungen und deren Ursachen.

Too many variables

Der Variablenbereich (ausgenommen Variablenfelder) überschreitet 64kByte.

Out of memory

Der Speicherplatz reicht für den Compiler nicht aus. Abhilfen:

RAM-Disk de-installieren (Rechner ausschalten)

Rechner mit mehr Speicher für den Compilier-Vorgang verwenden

Programm kürzen oder zerlegen und mit CHAIN Teile nachladen

Type mismatch

Sie übergeben einen falschen Wert an eine Prozedur oder Funktion (String statt Zahl oder umgekehrt)

Bad EXIT

EXIT geht im Compiler nur als EXIT ohne Anzahl oder als EXIT -1

EXIT 3 geht also z.B. nicht.

EXIT TO ist nur zum Verlassen einer Schleife erlaubt; EXIT TO kann nicht zum Verlassen einer Funktion oder Prozedur genutzt werden.

Structure too long

Eine Struktur (FOR..NEXT, REPEAT..UNTIL, WHILE..WEND-Schleifen, IF..THEN..ELSE) erstreckt sich über mehr als 32kByte Compilat.

Auch der Interpreter kann keine größeren Strukturen verwalten; das Compilat kann jedoch länger werden als das ursprüngliche BASIC-Programm.

Abhilfe: Teile des Schleifeninhalts als Prozeduren definieren.

Undefined statement(s) or DIMs

GOTOs, GOSUBs etc. zeigen auf Zeilen oder Labels, die nicht existieren.

Ein Variablenfeld muß DIMensioniert werden, bevor es benutzt wird. Im Interpreter ist dies nicht unbedingt notwendig; undefinierte Variablenfelder dimensioniert der Interpreter automatisch auf 10,10,10...

Abhilfe: Fügen Sie die entsprechenden DIM-Befehle am Programmanfang und hinter jedem CLEAR-Befehl ein.

9.1 WARNINGS

Die folgenden Meldungen des Compilers sind WARNUNGEN. Das heißt, sie führen nicht zwangsläufig zu einem nicht lauffähigen Compilat; Sie sollten sich die Warnungen aber genau ansehen, ob sie die Funktionssicherheit beeinträchtigen.

Warning: RETURN type mismatch

Eine selbstdefinierte mehrzeilige Funktion muß den Variablentyp zurückgeben, den sie als Postfix im Namen trägt.

Falsch: `DEF FN Addiere#(X,Y):RETURN X+Y`

Richtig: `DEF FN Addiere#(X#,Y#):RETURN X#+Y#`

Auch richtig: `DEF FN Addiere#(X,Y):RETURN CDBL(X+Y)`

Wenn eine Funktion als Double Float (Postfix #) definiert wurde, muß sie auch Double Float zurückgeben.

Warning: Unused statement(s):

Die aufgelisteten Prozeduren und Labels werden im gesamten Programm nicht benutzt. Sie können entfernt oder mit REM-Befehlen versehen werden.

9.2 Fehlermeldungen der Compilate

Normalerweise melden Basic-Compilate bei Auftreten eines Laufzeitfehlers genau die gleiche Fehlermeldung wie das interpretierte Programm (vergleiche Handbuch des Interpreters). Konnte jedoch die Fehlerzeile nicht ermittelt werden, so erscheint der Zusatz:

`USE "TRACE ON" TO GET ERRORLINE`

Dies weist darauf hin, daß um die Fehlerzeile festzustellen, das Compiler-Steuerwort "TRACE ON" eingefügt werden und das Programm erneut compilert werden muß. Ein eventuell vorhandenes Steuerwort "NO LINENUMBERS" muß natürlich ebenfalls entfernt werden (siehe auch im Kapitel Compiler-Steuerworte).

10. Der Segment Pointer

Die Systemvariable SEGPTR zeigt auf die folgende Pointertabelle (vgl. Funktion SEGPTR des Interpreters):

Offset	Speicherbereich
0	Zeilen-Nummern-Tabelle
4	Start des compilierten Codes
8	Ende der Labeltabelle
12	Start der Variablen
16	Reserviert
20	0 (siehe unten)
24	Start der Dateibuffer
28	0 (siehe unten)
32	Freier Bereich
36	festgehaltener Programmzeiger
40	Garbage Top (aktueller Zeiger)
44	Garbage Bottom
48	Garbage High (Ende des Garbage - höchste vom Basic belegte Adresse)
52	Stack-Maximalwert
56	-
60	Höchste Adresse des Prozessorstacks (SSP)
64	Adresse der Basepage
68.B	Flag für Mutitasking ("always" oder "between"
	Darf nur durch Compilersteuerworte verändert werden.
70	Beginn der ARRAYS (im Interpreter 20)
74	Beginn der Strings (im Interpreter 28)

Die Verlegung der letzten beiden Zeiger dient der Kompatibilität zum Interpreter: Die Ausdrücke

```
LPEEK(VARPTR(TEXT$))+ LPEEK(SEGPTR+20)
```

und

```
LPEEK(VARPTR(A(0,0)))+ LPEEK(SEGPTR+20)
```

können in genau dieser Form im Compiler verwendet werden. Der Inhalt des VARPTR schließt den Offset bereits mit ein, da der Compiler Strings und Arrays direkt addressiert. Der Ausdruck `LPEEK(SEGPTR + ..)` ist in diesen beiden Fällen null.

11. Speicheraufbau

Der Compiler unterstützt ab der Version 3.5 zwei verschiedene Speichermodelle: das **"dynamische"/Maximum-Speichermodell** und das **"statische"/Minimum-Speichermodell.**

Die Begriffe sind deshalb in Anführungszeichen gesetzt, da statisch nicht bedeutet, daß die dynamische String- und Arrayverwaltung nicht mehr funktionieren würden. Vielmehr meint statisch nur, daß der Gesamtbereich innerhalb dessen das Compilat Speicher belegt, sich nicht verändert.

11.1 Grundsätzlicher Speicheraufbau

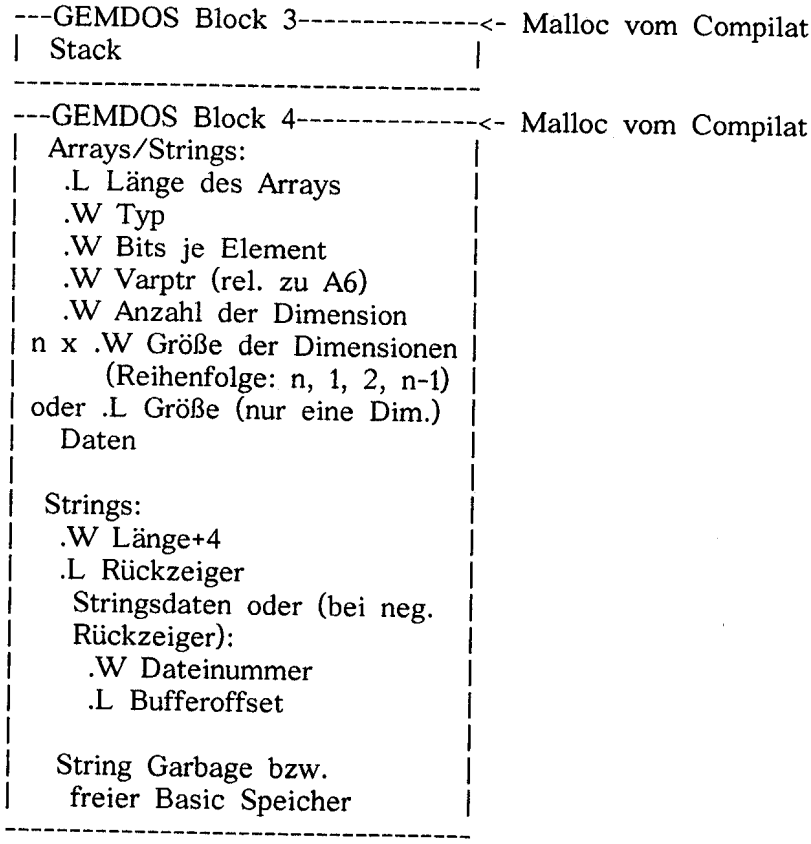
```
---GEMDOS Block 1-------------
|    Basepage                 |
-------------------------------  <- TEXT Segment
| BASLIB bzw. LOADER          |
-------------------------------  <- DATA Segment
| BC Tabelle:                 |  Offset zum "
|   .L Länge Programm         |    0
|   .L Länge Zeilen-Tabelle   |    4
|   .L Länge Label-Tabelle    |    8
|   .L Länge Reloc-Info       |    12
|   .L Länge #-Variablen      |    16
|   .L Länge !-Variablen      |    20
|   .L Länge %-Variablen      |    24
|   .L Länge $-Variablen      |    28
|   .L Länge Array-Pointer    |    32
|   .L Länge Library-Tabelle  |    36
|   .L Zeiger auf erstes DATA |    40
|   .L Intern verwendet       |    44
-------------------------------
| weiter auf der nächsten Seite |
```

```
| Fortsetzung der letzen Seite |
-----------------------------------
| Basic-Compilat:              |
|  BRA.S XX                    |   48
|  .L Reserviert               |   52
|  .L Reserviert               |   56
|  .L Stackgröße               |   60
|  .L Basic Speicher Größe     |   64
|                              |
| Hochoptimierter Code         |
|      "          "            |
-----------------------------------
| Zeilen-Tabelle:              |
|  .W Zeilennummer             |
|  .L relativer Zeiger         |
-----------------------------------
| Label-Tabelle:               |
|  8 Byte Labelname            |
|  .L relativer Zeiger         |
-----------------------------------
---GEMDOS Block 2 / BSS-------
| Einzelvariablen              |<- BSS (ab Version 3.5)
-----------------------------------
```

Die kursiv gesetzten Einträge gelten erst ab Version 3.5 und höher.

```
-----------------------------------   <- vom VDI angelegt,
| Daten von V_Opnvwk           |      falls notwendig
-----------------------------------
```

Dieser Block hat eigentlich überhaupt nichts mit dem Basic-Compilat zu tun und wird unter Umständen gar nicht erzeugt[1]. Er wird hier nur aufgeführt, weil er addressmäßig normalerweise zwischen dem 2. und dem 3. Block zu liegen kommt.



[1]Wenn das Compilat keinerlei AES- oder VDI Funktionen benutzt, wird nach dem Link-Vorgang auch kein V_Opnvwk mehr aufgerufen.

11.2 Das alte Speichermodell

Zum Vergleich soll nocheinmal kurz das alte Modell erläutert werden (Compilate 3.06 und früher):

1. GEMDOS Speicherblock (durch PEXEC erzeugt):

   Basepage

   TEXT-Segment: Loader bzw. BASLIB

   DATA-Segment: BC Tabelle, Compilat und Zusatzinfos

   BSS: nicht vorhanden

Dieser Block wird verkleinert bis Ende der Labeltabelle, d.h. vom DATA Segment ist die BC Tabelle, das Programm, die Zeilennummern Tabelle und die Label Tabelle noch da.

2. GEMDOS Speicherblock (per MALLOC angefordert):

   alle Einzelvariablen (einschließlich Arraypointer)

3. GEMDOS Speicherblock (per MALLOC angefordert):

   der Stack (Größe bestimmt durch CLEAR oder Default= 4096)

4. GEMDOS Speicherblock (per MALLOC angefordert):

   der dynamisch verwaltete Array- und Stringbereich (Garbage) Größe indirekt ebenfalls bestimmt durch CLEAR bzw. Default=Freier Speicher - 65536

Wie man sieht besteht jedes Compilat aus mindestens 4 Speicherblöcken. Außerdem ist der vierte Block zunächst (vom Startup-Code der BASLIB) immer auf Maximum reserviert und wird erst im eigentlichen Programm beim Ausführen des CLEAR wieder verkleinert. Dies führt speziell bei Accessories zu Problemen.

Wird außerdem noch ein CHAIN ausgeführt, so wird für das neue Compilat ein zusätzlicher Block angefordert. Vom ursprünglichen ersten Block bleibt nur noch das TEXT Segment (die BASLIB) und - wegen eines Bugs - die BC Tabelle bestehen. Für das neue DATA Segment des nachzuladenden Compilats wird ein zusätzlicher Block angelegt.

11.3 Das "dynamische" oder "Maximum"-Modell:

1. GEMDOS Speicherblock (durch PEXEC erzeugt):

Basepage

TEXT Segment: Loader bzw. BASLIB

DATA Segment: BC Tabelle, Compilat und Zusatzinfos

BSS: Platz für alle Einzelvariablen minus Länge Library Tabelle und Reloc-Info.

Dieser Speicherblock wird ganz normal über die drei Segmente verkleinert. Anschließend werden die Einzelvariablen am Ende des DATA Segments eingerichtet. Sie reichen dann bis ans Ende des BSS (der 2. Block entfällt).

3. GEMDOS Speicherblock (per MALLOC angefordert):

der Stack (Größe bestimmt durch CLEAR oder Default= 4096)

4. GEMDOS Speicherblock (per MALLOC angefordert):

der dynamisch verwaltete Array- und Stringbereich (Garbage) Größe indirekt ebenfalls bestimmt durch CLEAR bzw. Default=Freier Speicher - 65536

Der Vorteil ist, daß ein Speicherblock weniger angefordert werden muß. Der schwerwiegendere Nachteil bleibt jedoch: es wird immer noch zuerst der ganze Speicher belegt und dann wieder freigegeben.

Bei CHAIN kommt wie oben beschrieben ein weiterer Speicherblock hinzu.

11.4 Das "statische" oder "Minimum"-Modell:

1. GEMDOS Speicherblock (durch PEXEC erzeugt):

   Basepage

   TEXT Segment: Loader bzw. BASLIB

   DATA Segment: BC Tabelle, Compilat und Zusatzinfos

   BSS: Platz für alle Einzelvariablen, den Stack und den Array/Stringbereich minus Länge Library Tabelle und Reloc-Info.

Dieser Speicherblock wird wiederum ganz normal über die drei Segmente verkleinert. Anschließend werden die Einzelvariablen am Ende des DATA Segments eingerichtet. Es folgen direkt dahinter der Stack und der Array/Stringbereich. Die Blöcke 2-4 werden also allesamt im BSS zusammengefaßt.

Alle weiteren CLEAR Aufrufe mit Parametern bleiben unberücksichtigt, die Speicheraufteilung ändert sich nicht. **Dieses Speichermodell ist für Accessories vorgeschrieben.**

Bei CHAIN entsteht natürlich zwangsläufig trotzdem wieder ein zweiter Speicherblock, in den das neue Programm geladen wird.

Vorteil: nur ein einziger Speicherblock, dessen Grösse schon im Programmheader zu erkennen ist (wichtig beim Laden von Accessories). **Bei EXEC wird der Speicher nicht gelöscht, alle Variablen bleiben erhalten (gilt nur für das Compilat).**

Nachteil: es kann immer nur ein Minimum Modell sein, späteres Vergrössern ist ausgeschlossen. Dies wirkt sich z. B. beim Redimensionieren von Variablenfeldern aus: wenn das vergrößerte Feld nicht in den zuvor eingestellten Speicher paßt, wird ein Fehler erzeugt.

12. Geschwindigkeits-Optimierung

Das folgende Kapitel ist für diejenigen gedacht, die auf allerhöchste Geschwindigkeiten angewiesen sind.

12.1 Variablen-Typen

Überlegen Sie, ob Sie wirklich alles in Fließkomma rechnen müssen. In vielen Fällen reichen Integer-Zahlen aus. Long-Integer hat einen Zahlenbereich von -2147483648 bis 2147483647, und selbst wenn Sie zwei Stellen davon für feste Nachkommastellen reservieren, können Sie immer noch im 10-Millionen-Bereich rechnen.

12.2 Berechnungs-Typen

Bestimmte Rechenfunktionen bewirken automatisch, daß die gesamte Berechnung in Fließkomma durchgeführt wird (siehe Kapitel "Verarbeitungstypen der Funktionen"). Beispiel:

```
A=SQR(I)+J-I*3
```

Die Quadratwurzel liefert immer ein Fließkomma-Ergebnis, daher wird der Rest der Zeile auch in Fließkomma gerechnet (sonst könnten ja Nachkommastellen verloren gehen...).

Da das Ergebnis jedoch ohnehin nur in einer Integer-Variablen abgespeichert wird, ist die Fließkomma-Berechnung überflüssig.

Schneller:

```
A=SQR(I)+(J-I*3)
```

Jetzt wird die in der zweiten Klammer stehende Berechnung in Integer durchgeführt; erst anschließend wir das Ergebnis in Fließkomma gewandelt, und mit der Quadratwurzel zusammgengezählt.

Noch schneller:

```
A=CINT(SQR(I))+J-I*3
```

Mit CINT (Convert to Integer) verwandeln Sie den Fließkommawert, der beim Berechnen der Quadratwurzel entsteht, gleich in Integer.

Beachten Sie in diesem Zusammenhang, daß die normale Division ("/") Fließkomma erzwingt!! Berechnungen wie

```
A=A/5
```

sollten Sie unbedingt mit der Integer-Division "\" durchführen.

```
A=A\5
```

Manchmal kann es sich lohnen, eine Fließkomma-Konstante durch Integer-Brüche zu ersetzen:

`A=B*2.5` wird in Fließkomma gerechnet (98.7 Mikrosekunden)

`A=B*5\2` wird in Integer gerechnet (39.8 Mikrosekunden)

12.3 Konstanten

Fassen Sie Konstanten in Klammern zusammen. Zusammengefaßte Konstanten werden bereits beim Übersetzen berechnet - und nicht erst beim Abarbeiten des Programms.

Beispiel:

`A=B*(3*4)` wird automatisch zusammengefaßt zu `A=B*12`

Jedoch:

`A=B*3*4` wird nicht zusammengefaßt!

Denken Sie auch an's Umordnen:

A=J*2*K*5 läßt sich umordnen zu **A=J*K*(2*5)**

...und der Compiler kann optimieren!

12.4 Strings

Fassen Sie String-Berechnungen soweit wie möglich zusammen. Das Zwischenspeichern von Strings dauert relativ lange!

Statt:

```
A$="OMIKRON.COMPILER"
B$=LEFT$(A$,7)
C$=RIGHT$(A$,8)
D$=C$+B$
```

schreiben Sie:

```
A$="OMIKRON.COMPILER"
D$=RIGHT$(A$,8)+LEFT$(A$,7)
```

(natürlich nur, wenn Sie die Zwischenergebnisse nicht brauchen).

Das Zusammenfassen lohnt sich auch bei normalen Berechnungen; doch der Zeitgewinn ist bei weitem nicht so groß wie bei Strings.

12.5 IF..THEN

Vergleiche können vom Compiler besonders gut optimiert werden, wenn sie in folgender Form vorliegen:

```
IF A>B THEN...
```

A und B können auch beliebige Rechenausdrücke sein, ebenso kann es statt ">" natürlich auch "<", "=", ">=" usw. heißen, z.B.:

```
IF J*(3*4) = Anzahl(I)/2 THEN...
```

Weniger optimieren kann der Compiler bei Ausdrücken in der Art:

```
IF (A>B) AND (B<C) THEN...
```

Zur Beschleunigung schreibt man besser:

```
IF A>B THEN IF B<C THEN...
```

Auch geklammerte Vergleiche sind schlecht optimierbar:

```
IF (A>B) THEN...
```

ist langsamer als

```
IF A>B THEN...
```

Für Profis: Im ersten Fall handelt es sich um die Funktion "()", deren Argument ein Vergleich ist; im zweiten Fall handelt es sich um einen Vergleich.

12.6 Schleifenschachtelungen

Wenn möglich, nehmen Sie die häufiger durchlaufene Schleife als innere Schleife.

Langsamer:

```
FOR Art=1 TO Art_Anz
 FOR I=0 TO 1
  Preis(Art,I)=0
 NEXT I
NEXT Art
```

Schneller:

```
FOR I=0 TO 1
 FOR Art=1 TO Art_Anz
  Preis(Art,I)=0
 NEXT Art
NEXT I
```

13. Die mitgelieferten Programme

Auf der Diskette finden Sie die folgenden Programme:

COMPILER.PRG der eigentliche Compiler

CUTLIB.PRG: Dieses Programm bindet die BASLIB an ein Compilat an und kürzt die nicht benötigten Teile der BASLIB heraus. Wenn ein Programm also keinen Sinus berechnet, wird der entsprechende Teil der BASLIB auch nicht angebunden.

13.1 Bedienung von CUTLIB.PRG

13.1.1 Cutlib automatisch starten

Wenn Ihr Basicprogramm das Steuerwort **COMPILER "CUTLIB"** enthält, so wird nach erfolgreichem Compilieren Cutlib automatisch gestartet. Die BASLIB wird also immer gleich an das Basicprogramm angebunden und braucht nicht nachgeladen zu werden. (Siehe auch Kapitel "Compiler-Steuerworte")

13.1.2 Starten von CUTLIB.PRG vom Desktop

Starten Sie CUTLIB.PRG vom Desktop durch Doppelklick. Das Programm stellt eine Dateiauswahlbox dar, in der Sie das Programm anwählen können, an das Sie die BASLIB anbinden wollen.

Tip: Wie Besitzern älterer Compilerversionen vielleicht aufgefallen ist sind die Libraries erheblich länger geworden (> 70 kByte). Dies liegt daran, daß die zum Anbinden notwendige Information sich vollständig in der LIBRARY-Datei befindet (früher auf war diese Information auf zwei weitere Dateien verteilt: .REL und .REQ). Um Platz zu sparen können Sie sich auf folgende Weise eine wesentlich kürzere BASLIB erzeugen:

Schreiben Sie folgendes einzeiliges Basicprogramm:

```
COMPILER "CUTLIB -999"
```

compilieren Sie es und benennen das entstandene Programm in BASLIB35 um. Eine auf solche Weise erzeugte BASLIB ist allerdings nur noch zum Nachladen geeignet und kann von CUTLIB nicht mehr verwendet werden.

HILBERT.BAS: Ein Basicprogramm, das auch compiliert auf der Diskette verfügbar ist. Es zeigt den Geschwindigkeitsgewinn durch den Compiler besonders eindrucksvoll.

SHELL.BAS: Ein einfacher Kommandointerpreter, der auch Batch-Jobs verarbeitet.

SHELL.PRG: Dasselbe Programm in kompilierter Form.

Den Befehlsumfang der Shell können Sie aus dem Basicprogramm SHELL.BAS ersehen. Eine kurze Liste ist:

CHDIR, MKDIR, RMDIR, CLS, COPY, DATE, TIME, ERASE, DEL, DIR, ECHO, EXIT, RENAME, PAUSE, PROMPT, TYPE, REM, VER.

RAM200.PRG: Eine RESET-feste RAMDISK. Durch einfaches Umbenennen dieses Programms kann die Größe verändert werden: RAM1000.PRG ist eine 1000kByte große RAM-Disk, etc. Natürlich wird die Änderung erst beim nächsten Kaltstart wirksam.

ACC.BAS: Ein **Beispiel-Accessory**in Basic. Dieses Programm muß compiliert, geCUTLIB't und anschließend als ACC.ACC auf Laufwerk C: oder A: kopiert werden. Es läuft dann als Accessory; gibt aber, wenn es angeklickt wird, nur eine Alertbox aus.

ACC.BAS dient als Beispiel zur Programmierung von Accessories in OMIKRON.BASIC. Wenn Sie selber ein Accessory programmieren wollen, verwenden Sie am besten ACC.BAS und ersetzen die Zeile, in der die Alert Box aufgerufen wird, durch Ihr Programm.

Bitte wenden Sie sich mit allen Fragen zur Accessory-Programmierung nicht an uns, sondern ziehen Sie ein GEM-Buch zu Rate.

DEBUG.BAS und **DEBUG.PRG**: Ein Beispiel zur Verwendung eines Debuggers zusammen mit OMIKRON.Basic Compilaten. Das Programm ist ausführlich kommentiert und somit weitgehend selbsterklärend.

14. Glossar

Accessory

Hintergrundprogramm, welches unter GEM quasi gleichzeitig mit der Hauptapplikation laufen kann. Accessories werden - anders als normale Programme - bereits kurz nach dem Einschalten des Rechner in den Hauptspeicher geladen. Die Art des Ladens macht eine besondere Art der Speicherbenutzung erforderlich.

BIOS

Abkürzung für Basic Input Output System. Ein Teil des Betriebsystems, das sich um grundlegende Ein- und Ausgabeoperationen kümmert (z.B. Zeichen Ein- und Ausgabe auf dem Bildschirm bzw. Tastatur)

Basepage

Eine vom Betriebsystem für jedes Programm angelegte Kontrollstruktur, die Informationen über das Programm enthält.

Compilat

Ein direkt ausführbares Programm, welches durch einen Compiler automatisch erzeugt wurde.

Compiler

Automatisches Übersetzerprogramm, der ein in einer höheren Programmiersprache geschriebenes Programm in ein direkt ausführbares Maschinenprogramm übersetzt.

Compiler-Library (BASLIB)

Eine Sammlung von Unterprogrammen, die von jedem compilierten Basicprogramm benötigt werden.

Coprozessor-FPU

Abkürzung für Coprozessor Floating Point Unit. Ein direkt mit dem Hauptprozessor verbundener Zusatzprozessor, der bestimmte Fließkommaoperationen stark beschleunigt. Im Unterschied zur IO-FPU ist eine direkte Zusammenarbeit mit dem Hauptprozessor möglich. Fließkommabefehle werden direkt ausgeführt. Ein Coprozessor dieser Bauform findet sich z.B. im ATARI TT oder auf manchen Beschleunigerkarten. Er wird vom Compiler 3.5 nicht unterstützt, dafür brauchen Sie das TT-Paket.

Debugger

Spezielles Programm zur Fehlersuche. Mit Hilfe eines Debuggers können Maschinenprogramme im Einzelschrittmodus ausgeführt und Schritt für Schritt genau durchleuchtet werden.

Default(Einstellung)

Voreinstellungen, die immer dann Verwendung finden, wenn keine anders lautenden Angaben gemacht werden.

Editor

Programm zum Bearbeiten von Texten oder Programmen. Speziell beim OMIKRON.Basic ist der Editor in den Interpreter intergriert. Man kann also im Wechsel das Programm ausführen, abändern und wieder ausführen.

Exception Handler

Spezielles Unterprogramm, das zur Behandlung eines Ausnahmefalles dient. Die Motorola-Prozessoren 680X0 erzeugen in bestimmten Fällen (z.B. einer Division durch Null) eine sogenannte Ausnahme. Für jeden solcher Ausnahmefälle ist nun ein Exception Handler zuständig, der festlegt, wie weiter verfahren werden soll (z.B. Fehler ausgeben und Programm beenden).

Extension siehe Namenserweiterung

FPU 68881

Spezieller Fließkomma Coprozessor, der zusammen mit den Motorola Prozessoren 680X0 eingesetzt werden kann. In Verbindung mit einem 68020 oder 68030 ist sogar ein echter Coprozessor Einsatz möglich (d.h. Fließkommabefehle können direkt verarbeitet werden). Dagegen kann die FPU zusammen mit dem 68000 nur als Ein- Ausgabegerät angesprochen werden (IO-FPU), dem jeder Befehl und jeder Zahlenwert einzeln übermittelt werden müssen.

GEMDOS

Teil des Betriebsystems, der sich auf übergeordneter Ebene um die Verwaltung der Resourcen des Rechners kümmert. Die Verwaltung des gesamten Dateisystems, einschließlich Inhaltsverzeichnissen und die Vergabe von Speicher zählt zu den Aufgaben des GEMDOS.

Garbage (Speicher) bzw. String-Garbage

Ein besonders schnelle Art der Speicherverwaltung, die vor allem bei ständig wechseldem Platzbedarf (z.B. bei Zeichenketten) sehr effizient arbeitet. Durch die Art der Neuvergabe von Speicher wird allerdings ständig Müll (Garbage) produziert, d.h. Speicherstücke, die nicht mehr benutzt werden können. Es ist deshalb notwendig, einen solchen Speicher von Zeit zu Zeit aufzuräumen (Garbage Collection).

Hauptapplikation

Unter GEM ist es möglich, mehrere Programme quasi gleichzeitig ablaufen zu lassen. Eines davon ist die im Vordergrund laufende Hauptapplikation, daneben sind noch bis zu sechs Accessories möglich. Eine Applikation ist also nichts anderes als ein unter GEM laufendes Programm (mit Menüzeile, Dialogboxen ...).

Hochsprache

Sogenannte "höhere" Programmiersprache, d.h. es ist ein hoher Abstraktionsgrad gegenüber der zu programmierenden Maschine erreicht. Gegensatz: Assemblersprachen orientieren sich direkt am zu Grunde liegenden Prozessor und sind speziell auf diesen zugeschnitten.

Initialisieren

Vorbelegen bestimmter Speicherbereiche mit Standardwerten (z.B. Null).

IO-FPU

Fließkomma-Coprozessor, der wie ein Ein- Ausgabegerät angesprochen wird. Alle Befehle und Werte müssen einzeln zum Coprozessor übertragen und von dort wieder abgeholt werden. Dies braucht im Gegensatz zur Verwendung als echte Coprozessor-FPU (wie z.B. im ATARI TT) sehr viel Zeit, obwohl natürlich gegenüber der normalen Rechenzeit eines Einzelprozessors noch sehr viel gut gemacht wird. Mit einem Motorola 68000 Prozessor ist nur die Verwendung als IO-FPU möglich, da erst ab den Prozessortypen 68020 und 68030 eine direkte Verbindung zur FPU vorgesehen ist.

Interpreter

Ein Interpreter arbeitet ein Programm Befehl für Befehl ab, wobei während der Laufzeit des Programms die einzelen Befehle in entsprechende Aktionen umgesetzt werden. Dieses schrittweise Abarbeiten ist langsamer als ein einmaliges Vorübersetzen und anschließendes Direktausführen, hat aber den Vorteil, daß nach geringfügigen Änderungen das Programm sofort wieder gestartet werden kann. Wäh-

rend der Entwicklungsphase ist also der Interpreter ein geeignetes Werkzeug zum Testen, wohingegen am Ende immer ein fertig übersetztes Programm stehen sollte.

Laufwerk

Der Laufwerkname ist ein Name von "A:" bis "P:". Das große "A" steht für die Diskette, die Buchstaben von "C" aufwärts stehen meist für die einzelnen Abteilungen einer Festplatte.

Library (BASLIB)

Eine Sammlung von Unterprogrammen, die von jedem compilierten Basicprogramm benötigt werden.

Library-Tabelle

Tabelle von binären Einträgen, wo festgelegt ist, ob ein bestimmtes Unterprogramm der BASLIB benötigt wird oder nicht.

Linker

Dienstprogramm, das die Library (BASLIB) an ein fertig compilertes Basicprogramm anbindet (engl. to link = verbinden).

MALLOC

Betriebsystemfunktion zur Reservierung von Speicher (Memory Allocate).

Maschinensprache

Für die Maschine (den Prozessor) direkt umsetzbare Folge von Befehlen, die in binär kodierter Form vorliegen.

Multitasking

Quasi gleichzeitiges Ausführen mehrerer Programme oder Unterprogramme. Beispiel: Ausgabe von Listen auf dem Drucker und gleichzeitige Dateneingabe am Bilschirm.

Namenserweiterung/Extension/Endung

Der letzte Teil des Dateinamens der durch Punkt abgetrennt wird heißt Extension. Die Extension zeigt an, um was für eine Art von Datei es sich handelt.

Übliche Extensions sind:

.DOC oder .TXT für Texte, Anleitungen

.PIC für Bilder (engl.: picture)

.PRG, .TOS oder .TTP für Programme

.INF für Einstellungs-Dateien (interne Information für ein Programm)

.BAS für BASIC-Programme

Offset

Abstand oder Versatz zu einer bestimmten Speicheradresse.

Pfad

Der Pfad ist die vollständige Bezeichnung einer Datei und besteht aus drei Teilen:

Der Laufwerkname: A:

Der Pfadname: \AUTO\

Der Dateiname: ADRESSEN.DAT

Der Laufwerkname ist ein Name von "A:" bis "P:". Das große "A" steht für die Diskette, die Buchstaben von "C" aufwärts stehen meist für die einzelnen Abteilungen einer Festplatte. Der Rückwärts \ Schrägstrich trennt diese drei Teile voneinander. Der eigentliche Dateiname besteht (genauso wie Ordnernamen) aus bis zu acht Zeichen (ADRESSEN), einem Punkt und dann einer Endung (Extension) aus bis zu drei weiteren Zeichen (.DAT).

Bei der Suche nach Dateinamen können auch sogenannte Joker-Zeichen verwendet werden:

Das Fragezeichen ? kann ein einzelnes beliebiges Zeichen ersetzen. Das Sternchen kann den Rest des Dateinamens oder der Endung (also mehrere Zeichen) ersetzen.

Beispiele:

A:\AUTO*.PRG bezeichnet alle Programme (Endung: .PRG) im AUTO-Ordner.

A:\AUTO\MORTIMER.* bezeichnet sowohl MORTIMER.PRG als auch MORTIMER.INF.

A:\TE?T bezeichnet sowohl die Datei TEST als auch die Datei TEXT. Anstelle des Fragezeichens kann also jeder beliebige einzelne Buchstabe stehen. Im Laufwerksnamen und in den Ordnernamen des Pfades sind Joker-Zeichen übrigens nicht zulässig.

Quelltext

Programmtext, meist in einer höheren Programmiersprache geschrieben, der, im Gegensatz zur fertig übersetzten Version, noch einsehbar und änderbar ist.

RAMDISK

Ein im Hauptspeicher des Rechners simuliertes Disklaufwerk. Vorteil: alle Schreib- und Leseoperationen gehen sehr schnell vonstatten. Nachteil: beim Ausschalten des Rechners sind die Daten verloren. Das Multiutility MORTIMER enthält beispielsweise eine sehr komfortable Ramdisk.

Reloc-Info

Tabelle aller Speicher-Adressen, die nach dem Laden des Programms reloziert werden müssen. Da man beim Erstellen eines Programms nicht weiß, an welcher Stelle des

Speichers es später ausgeführt wird, müssen nach dem Laden alle absoluten Adressen angepaßt werden. Diesen Vorgang nennt man Relozieren.

Service-Routine

Unterprogramm, das sich um die Behandlung einer Unterbrechung kümmert. Wird z.B. von der Tastatur eine Unterbrechung ausgelöst, so sorgt die Service-Routine davor, daß von dort eine Taste abgeholt und in einen Zwischenspeicher geschrieben wird.

Stapel bzw. Stack, Prozessor-

Speicher zum Zwischensichern wichtiger Adressen oder Registerinhalte (z.B. Rücksprungadresse beim Unterprogrammaufruf). Der Stack arbeitet nach dem LIFO-Prinzip (was zuletzt hinein geht, kommt zuerst wieder heraus).

Startup-Code

Programmteil eines jeden Compilats, der bestimmte Initialisierungen vornimmt wie z.B. Speicher reservieren, Cursor einschalten usw. Dieser Programmteil wird beim Starten eines Compilats immer zuallererst ausgeführt, noch vor dem eigentlichen Basicprogramm.

Symboltabelle

Verzeichnis aller in einem Programm verwendeten Namen (für Prozeduren oder Funktionen). Eine Symboltabelle ist Teil der Programmdatei und kann von einem Debugger geladen und angezeigt werden (symbolisches Debuggen).

TEXT, DATA und BSS

Normalerweise sind ausführbare Programme in drei Segmente unterteilt: das Text-, das Daten- und das BS(Block Storage)-Segment. Diese Unterteilung hat einen organisatorischen Hintergrund. Die meisten Compilersprachen, die mit Linkern arbeiten, setzen eine solche Dreiteilung vor-

aus, um mehrere Module verbinden zu können. Unter OMIKRON.Basic spielt diese Aufteilung eigentlich keine Rolle.

Unterbrechung/Unterbrechungsanforderung/"Interrupt"

Eine Unterbrechungsanforderung wird immer durch ein bestimmtes Ereignis ausgelöst (z.B. das Drücken einer Taste oder das Ablaufen eines Zeitgebers). Der Prozessor versucht dann sobald wie möglich in ein entsprechendes Behandlungsunterprogramm zu verzweigen.

Verzeichnis/Ordner/Unterverzeichnis

Verschiedene Namen für ein Inhaltsverzeichnis einer Diskette oderFestplatte. In einem Ordner können sich viele einzelne Dateien(Texte, Bilder, Programme), aber auch andere Ordner befinden. Wenn Sie einen Ordner 'öffnen', sehen Sie das Verzeichnis seines Inhaltes.

XBIOS

Abkürzung für eXtended Basic Input Output System. Ein Teil des Betriebssystems, der sich um erweiterte Ein- oder Ausgabemöglichkeiten des Rechners kümmert. (z.B. Mauseinstellungen oder Bildschirm ausdrucken)

Zeiger bzw. Pointer

Ein Verweis auf eine Variable oder eine Struktur. Meist ist ein solcher Verweis einfach durch Angabe der entsprechenden Speicheradresse verwirklicht. Wenn man also den Zeiger auf eine Struktur kennt, so kennt man deren Adresse und kann so auf die Struktur zu greifen.

relativer Zeiger

Ein Verweis wie oben allerdings muß noch eine Basisadresse eines Speichersegments hinzugezählt werden. Der relative Zeiger gibt also nur den Abstand zum Segmentanfang an.

15. Index

S

T

U

V

W

X

Z